AIAA-2000-4390

MANAGING THE PLAYBACK AND RECORDING OF SIMULATION MODELS IN AN OBJECT-ORIENTED SIMULATION

P. Sean Kenney*

Systems Development Branch
NASA Langley Research Center
Mail Stop 125B
Hampton VA 23681

Paul C. Sugden†

Unisys Corporation
NASA Langley Research Center
Mail Stop 169
Hampton, VA 23681

Abstract

Flight simulation often uses playback as a means to replay the movement of vehicles in a simulation environment. Playback vehicles must be absolutely repeatable and computationally efficient. This paper presents an object-oriented design for playing back vehicle behavior using a direct vehicle-state playback, control-surface playback, or pilot input playback. The design was added to the Langley Standard Simulation in C++ (LaSRS++) where it supports a number of current experiments. The design provides an accurate means to replay the dynamic behavior of a vehicle while minimizing computational load. The design has proven to be flexible, maintainable, and extendible.

Introduction

Among the many things required to provide a repeatable flight environment, one of the more complex items is the presentation of other vehicles to a pilot. In military aircraft simulations a pilot may have to acquire and destroy drones, maneuvering aircraft, fixed-based ground targets, or moving ground targets. In commercial aircraft simulations a pilot may have to deal with air and ground traffic while flying or taxiing in an airport's vicinity. Qualitative research data can only be obtained if the vehicles in these scenarios move with absolute repeatability.

Another complex requirement is to provide the means to replay the flight profile of a vehicle. Flight data from an aircraft that has crashed can be analyzed by re-creating the flight. Simulation models can also be verified against test flight data. Analysis is difficult without a simple, repeatable method of injecting data into a simulation model.

---

* Aerospace Engineer, Member AIAA.

† Software Engineer

The Langley Standard Real-Time Simulation in C++ (LaSRS++) provides this repeatability using three different methods: state playback, control-surface playback, and pilot playback[1]. State playback records the states of a vehicle and uses the data to "playback" the vehicle's movement without performing any integration. Control-surface playback records the control surface commands or deflections and uses the data to recreate the dynamic movement of a vehicle by driving control surfaces with the commands or deflections and integrating the state of the vehicle. Pilot playback records the inputs made by a pilot and uses the data to playback the flight by injecting the recorded pilot commands into the fully operational simulation. Each of the three methods is applicable to different kinds of simulation research.

State playback is applicable whenever pilot interaction with another vehicle is only dependent upon that vehicle's position, velocity, and orientation. For example, a fighter pilot might be tasked to target and destroy another aircraft. To accomplish the task, the pilot must be able to see the CGI representation of the target, track the target with radar, and use a gun sight to destroy the target aircraft. The CGI model, radar model, and gun model only need the target's states to compute the its position, velocity and orientation relative to the pilot. Derived states for an aircraft like angle of attack, Mach number, and sideslip are not needed. Thus, state playback computational load is light when compared to a complete six degree of freedom simulation model. This allows multiple state-playback models to be used within the time constraints of a real-time simulation frame.

Control-surface playback supports analysis of the simulation that requires removal of control law influences. For example, surface commands or surface deflections can be obtained from flight test data to recreate the flight profile of the test vehicle. The performance of the simulation model can then be validated against the flight profile of the test vehicle. Control-surface playback isolates the performance of a model from its control system, thereby allowing a developer to measure the performance of modifications to the aerodynamic package. Moreover, the developer can test the aerodynamics package before the control law has been completed.

Pilot playback is used to analyze the response of a human to his/her environment or the response of a simulation model to a series of human inputs. For example, if a pilot is tasked to perform emergency maneuvers as part of a training exercise, the pilot's inputs can be replayed while the instructor discusses the pilot's performance. State-playback would not provide all of the derived information necessary to completely re-create the test sequence as viewed by the pilot. Another application of pilot-playback is to analyze any unexpected behavior produced by a simulation model. If a pilot encounters an unexpected event while flying, the recorded inputs can be used offline to repeat the scenario so that the problem may be analyzed without requiring all of the resources necessary to perform pilot-in-the-loop, synchronous real-time simulations. Pilot playback also provides an ideal means to generate a demonstration of a simulation facility without a pilot in the loop thus allowing visitors an unobstructed view of the facilities.

Design Requirements

An object-oriented design was created to allow the playback and recording of dynamic vehicle data in a simulation framework. The design had the following requirements:

1. The design must provide several general capabilities that may be incorporated into any type of playback.
    a. The design must allow a vehicle to specify its starting point within the playback file.
    b. The design should have a minimal impact on the performance of the framework when there are vehicles in playback mode and no impact on performance when there are no vehicles in playback mode.
    c. The design must provide a method to delay the start of playback for a specified amount of time.

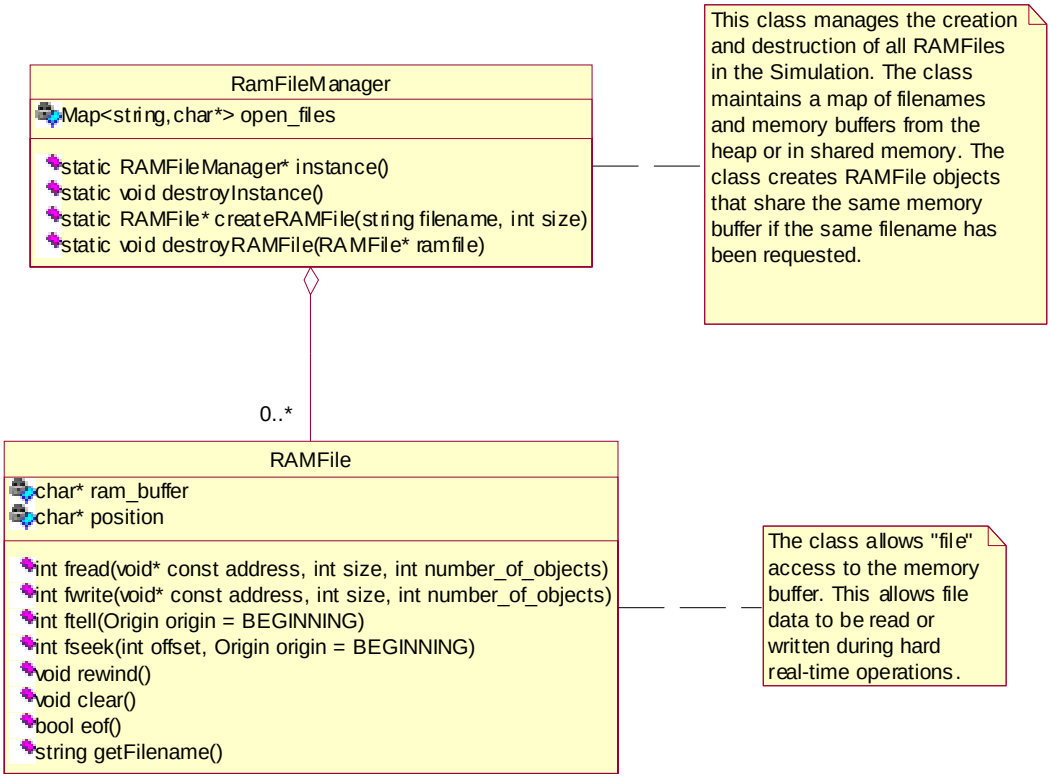

Figure 1 – RAMFile and RAMFileManager

d. The design must provide a means to continuously loop.
e. The design must provide a mechanism for recording data at integral divisions of the simulation frequency.
f. The design must provide a mechanism for playing back data recorded with a different frequency than the simulation.
g. The design must allow multiple vehicles to be driven from one playback file.

2. The design should provide several specific playback capabilities.
   a. The design should support an airport traffic mode where the playback vehicles have some intelligence when taxiing. A vehicle should not run into other vehicles, should not incur upon the runway when another aircraft is taking off, should stop smoothly when forced to halt, etc…
   b. The design should support the playback of surface deflections or surface commands.
   c. The design should support the playback of cockpit inputs.

A design that met the above requirements was implemented in the Langley Standard Real-Time Simulation in C++ (LaSRS++) application framework. LaSRS++ provides a powerful object-oriented framework for dynamic vehicle simulation in real-time[3]. The framework's object-oriented design makes the software extremely flexible, easily maintainable, and provides a high degree of re-use[4]. Encapsulation was used to hide implementation details. Inheritance and aggregation of common subcomponents were used to achieve maximal code reuse. Virtual method interfaces were utilized to obtain the advantages of polymorphism. Documented design patterns were used where possible[2]. The Builder and Singleton patterns were used in the design. The Builder pattern separates the construction of a

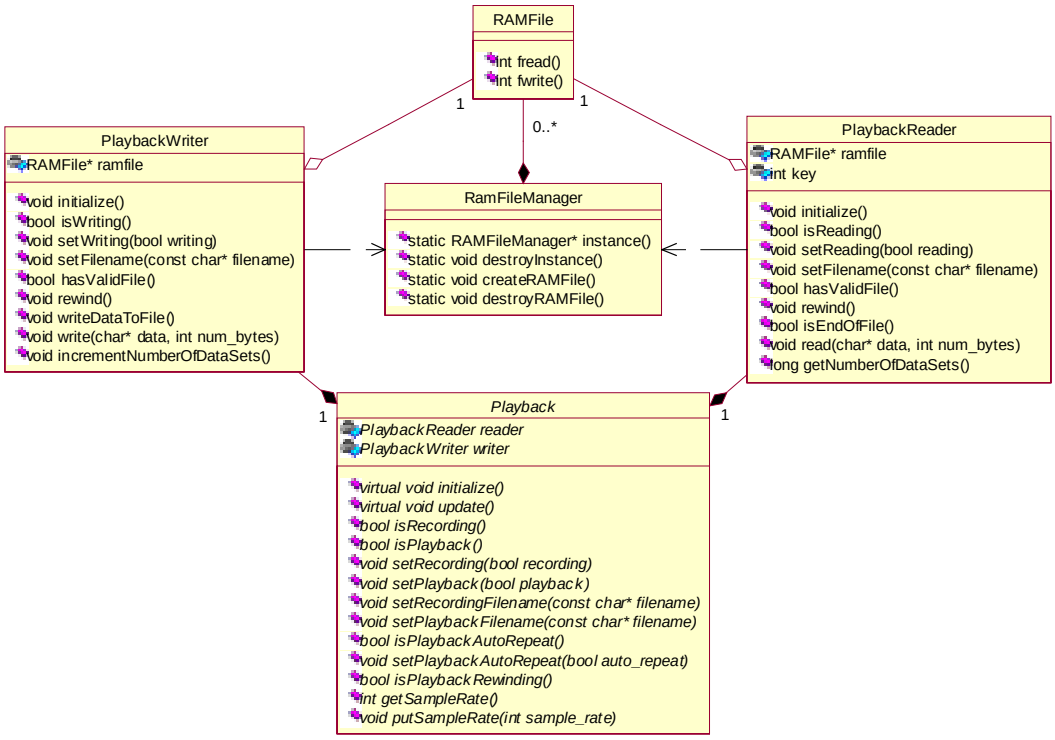

Figure 2 – Playback

complex object from its representation so that the same construction process can create different representations. The Singleton pattern ensures only one instance of a class exists and provides a global point of access to it.

RAMFile

The core of the design is a class called *RAMFile*. This class allocates a block of memory on the heap or in shared memory, and provides an interface that allows a client to treat the block of memory like a UNIX file. A client can read or write the "file" without the performance penalty associated with actual file access. This allows a client to write data to the *RAMFile* during hard real-time operations and then copy that data from the *RAMFile* to physical disk when the constraints of hard real-time are no longer in effect. The class also allows the file to be accessed by more than one client at a time. Figure 1 uses the Unified Modeling Language (UML) to show the class diagram for the *RAMFile* class and its relationship to a second class named *RAMFileManager*.

RAMFileManager manages the creation and destruction of all *RAMFile* objects in the simulation. The Singleton pattern ensures that only one instance of this class is ever created. *RAMFileManager* fields requests by clients to create *RAMFile* objects via the *createRAMFile* method. The method returns a *RAMFile* reference. If a client is the first client to request that the contents of a file be used in a *RAMFile*, *RAMFileManager* allocates a memory buffer and loads the file's contents into the buffer. The *RAMFileManager* then passes the buffer to the *RAMFile* when it is instantiated. If a client asks for a file that has already been loaded into a buffer for use by another client, the *RAMFileManager* class instantiates a *RAMFile* object by passing the pre-existing memory buffer to *RAMFile*'s constructor. This allows multiple clients to work on a single file without complicating the interface of *RAMFile*. Similarly, each client calls *destroyRAMFile* when the client is finished

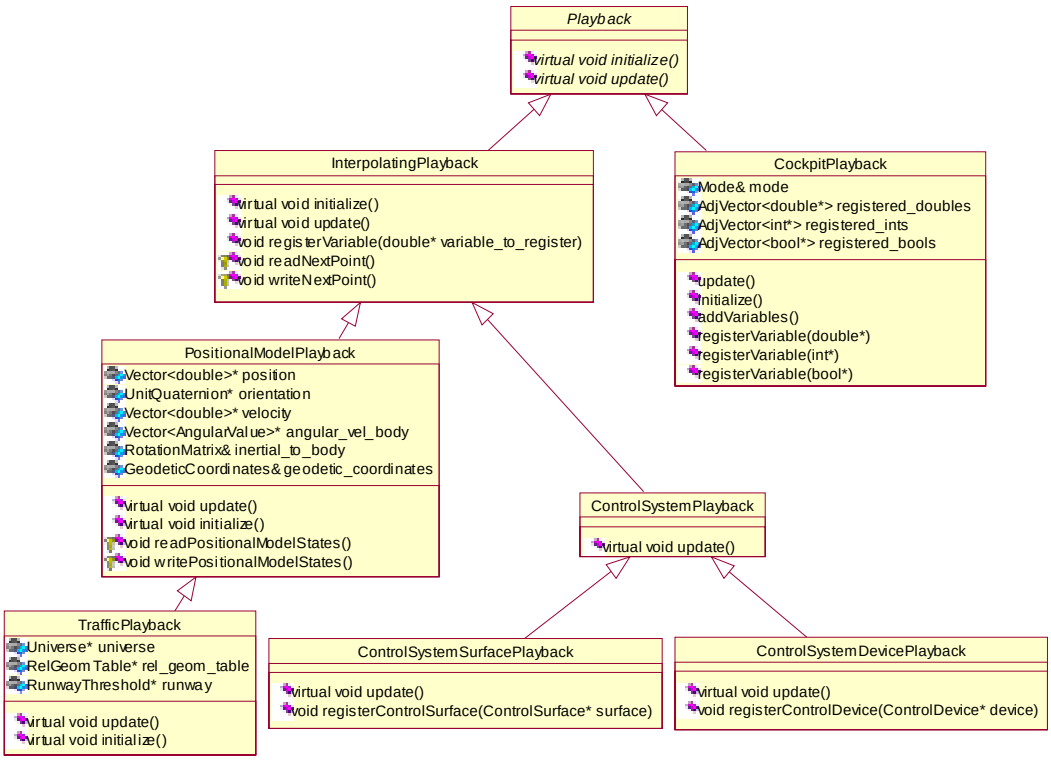

Figure 3 – The Playback Hierarchy

using a particular file. *RAMFileManager* only destroys the memory buffer for a file when there are no longer any *RAMFile* objects referencing the buffer.

Playback

Playback is an abstract class intended to be the base class for all types of playback operations. The class was designed to provide a generic interface for the recording and playback of data. Figure 2 demonstrates the Playback class and its dependencies.

Playback has two virtual methods that must be defined by all concrete derived classes. The *initialize* method is expected to initialize a derived class before or after data has been recorded or played back. Transfers to/from physical disk should occur when this method is called. The *update* method is intended to actually transfer data between the *RAMFile* and the simulation models. All of the other methods defined in Playback are to be used by clients and derived classes to manage the two classes contained within *Playback*, *PlaybackReader* and *PlaybackWriter*.

PlaybackReader provides an interface for reading playback data. If a valid filename has been given and the *setReading* method has been called with *true* as its argument, the class requests *RAMFileManager* to create a *RAMFile* for the file and then copies the data into the RAMFile if it is the first client of the *RAMFile*. If it is not the first client of this *RAMFile* then the data has already been loaded into memory from the file. Aggregation of a *RAMFile* object rather than inheritance allows *PlaybackReader* to hide the subset of *RAMFile*'s interface for modifying the file, while providing wrapper methods allowing reuse of *RAMFile*'s reading capabilities. The *PlaybackReader* class can now be used to read the data.

PlaybackWriter works in a similar manner except it writes data into the *RAMFile* and then copies it to physical file when *initialize* is called. The class assumes that it is the only user of a *RAMFile* because it is writing data. In a manner similar to *PlaybackReader*, *PlaybackWriter* prevents clients from accessing the reading capabilities of *RAMFile*'s interface

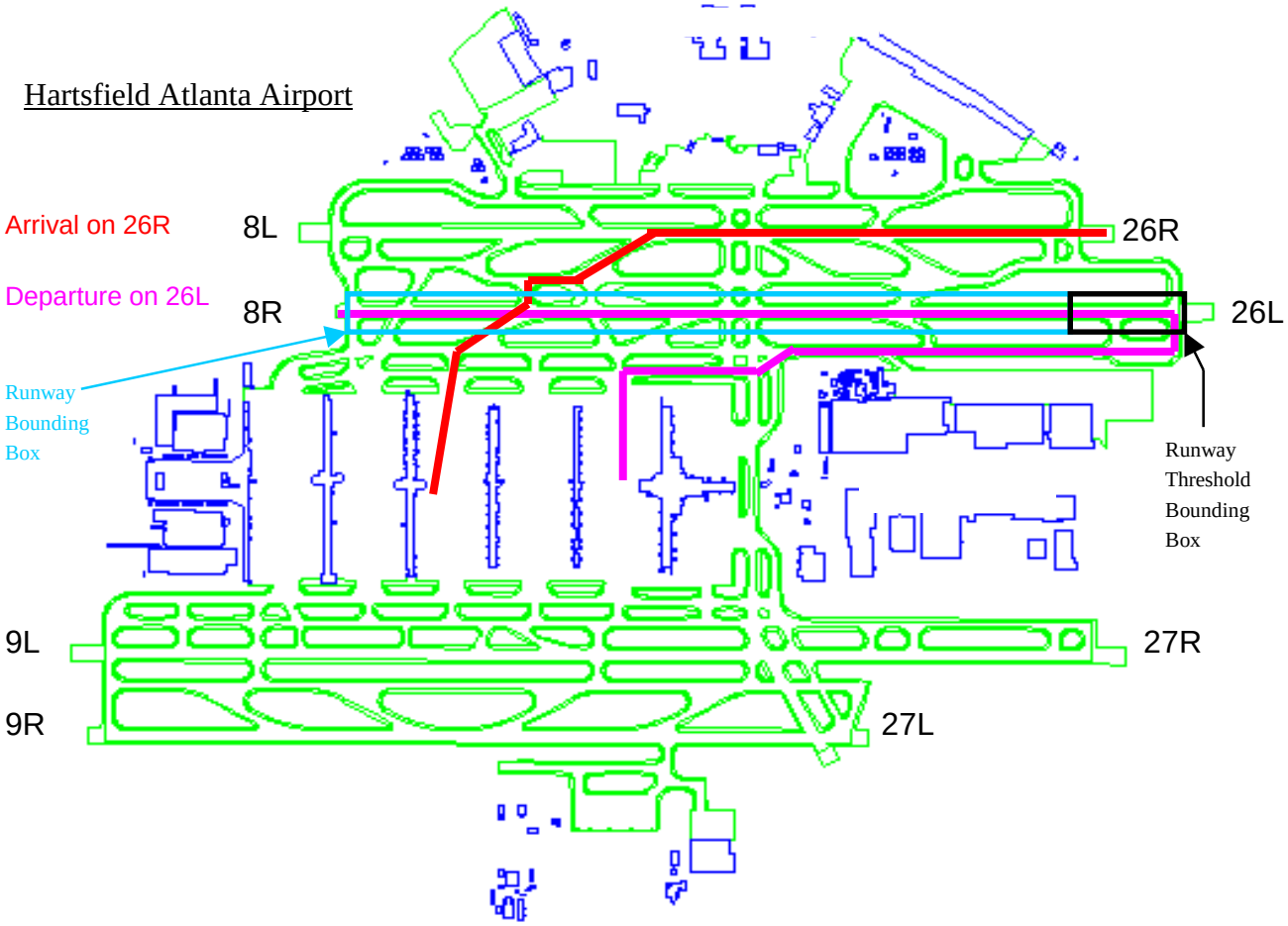

Figure 4 – Airport Traffic Diagram

while providing access to *RAMFile*'s writing capabilities through wrapper methods.

Interpolating Playback

InterpolatingPlayback is a concrete class that records data at a fixed integral division of the simulation frequency while allowing data to be played back every frame by interpolating between the sampled data points. Figure 3 illustrates that InterpolatingPlayback inherits from Playback and that the class implements the *initialize* and *update* virtual methods declared in the interface of *Playback*. The class also uses all of the methods defined in *Playback* to manage the recording and playback of files.

The method *registerVariable* allows client classes to register references to variables that are to be recorded and played back. Each variable that is registered with InterpolatingPlayback is added to a vector of variables. When recording, the method *writeNextPoint* first writes the current time to the *RAMFile* and then iterates through the vector of registered variables writing the value of each variable to the *RAMFile*. The *update* method makes sure that the *writeNextPoint* method is only called at the selected recording frequency. During playback, when the *update* method is called r*eadNextPoint* is called to read data sets from the playback file until it finds one which occurs after the current simulation time. The method *readNextPoint* saves this set and the previous one. A linear interpolation between these two sets determines the current set. These calculated values are then copied into the registered variables. If the recording frequency equals the playback frequency, no interpolation will be performed and

the played back data will be identical to the recorded data.

PositionalModelPlayback

In the LaSRS++ framework, a *PositionalModel* is a simulation object that has a position, orientation, translational velocity, angular velocity, and a set of geodetic coordinates. These are the minimum attributes required by the framework for a simulation object to be represented visually using a CGI and to support the computation of relative geometry between simulation objects. The *Vehicle* class extends *PositionalModel* by inheriting from it and adding acceleration as a state. *Aircraft* in turn, extends *Vehicle* by adding all of the states of an aircraft (angle of attack, sideslip, etc…). Since all vehicles in the LaSRS++ framework are descendants of *PositionalModel*, the motion of any vehicle may be recorded and played back by recording the states of a *PositionalModel*. The *PositionalModelPlayback* class provides a means to record and playback these states.

The constructor of *PositionalModelPlayback* requires references to the state vectors of *PositionalModel*. This is possible because *PositionalModelPlayback* is contained within *PositionalModel*. *PositionalModelPlayback* inherits from *InterpolatingPlayback* and uses the *registerVariable* method to register each component of the states. When the *update* method is called, the class uses the *update* method in *InterpolatingPlayback* to record or playback the state data. If playing back data, the method then computes several derived quantities needed by the simulation framework.

TrafficPlayback

TrafficPlayback is a class that extends *PositionalModelPlayback* by providing logic to make a playback positional model behave like aircraft traffic around an airport. The logic assumes that all traffic is centered about a pair of parallel runways and that the arriving traffic must cross the departure runway. The logic also assumes that the behavior of a recorded aircraft is consistent with that of an aircraft operating at an airport. Figure 4 shows a typical airport layout where the *TrafficPlayback* class can be used. On this diagram all departing aircraft are leaving on runway 26L and all arriving aircraft are on 26R. The arriving aircraft must cross the departure runway to get to the terminal.

The class performs several different computations to ensure that a playback behaves like an aircraft while taxiing. First, it uses the relative geometry information computed by the framework to ensure that the model is not going to collide with another model. If a collision is predicted, then the simulation model is brought to a halt until the path is clear again. Next, the class manages whether a model may begin to cross or enter the departure runway. To do this the class preprocesses the playback file when the class is first constructed and then determines on which frames the playback model would enter or exit the departure runway. The departure runway is broken into two zones: the runway threshold bounding box and the runway bounding box as illustrated on figure 4. By tracking when a model is entering either of the two zones, the class can manage whether or not the model can enter either area because it also knows whether another model is active in either part of the runway. For example, if a departing aircraft has already moved onto the threshold and is ready for takeoff, any arriving aircraft that is already crossing the departure runway on its way to the terminal would be allowed to continue. The departure aircraft is not allowed to begin its takeoff until after the runway is clear. But any arriving aircraft that are approaching the departure runway are forced to stop at the hold short line (the outer edge of each bounding box) so that the departing aircraft can take off. Once the departing aircraft is airborne the arriving aircraft is allowed to continue.

Finally the class gives the appearance of a continuously busy airport by rewinding a file when the playback has reached its end and then starting over. A departure playback file, for example, starts at a terminal and taxis to the departure runway. After takeoff the aircraft flies for several more minutes. By repeating the file again, one aircraft can be used over and over to give the appearance of a busy airport. The class also allows the simulation user to specify a starting location in the playback file by specifying a latitude, longitude and altitude. This allows multiple models to use the same playback file but start at different time locations

within the file. The class also allows the simulation user to specify the amount of time the model should pause before starting playback. This option allows the simulation operator to subject a pilot to traffic events such as near collisions, runway incursions, and other unusual circumstances.

ControlSystemPlayback

In the LaSRS++ framework, a *ControlSystem* maintains a collection of *ControlSurfaces*. A *ControlSurface* has a position command and a deflection. In closed loop simulation, control laws transform pilot inputs into surface position commands. The particular control surface then transforms the commanded position into a deflection.

ControlSystemPlayback is derived from *InterpolatingPlayback* and serves as a base for recording and playback of data required for repeatability during playback of control surface commands or deflections. Since the behavior of an aircraft is dependent both on the control surfaces and the engine thrust, it is necessary to record and play back the throttles so that playback of the control system results in a model with identical behavior to the recorded model. *ControlSystemPlayback* overrides the virtual *update* method of *InterpolatingPlayback* to transparently record and playback throttle positions. It is left as the responsibility of child classes to record and playback surface commands or positions.

The relationship between *ControlSystem* and *ControlSystemPlayback* is analogous to that between *PositionalModel* and *PositionalModelPlayback*. *ControlSystemPlayback* objects are constructed by and contained within *ControlSystem* objects. It is the responsibility of the *ControlSystem* object to expose its internal state to the playback object. The mechanisms by which the *ControlSystem* exposes its state are defined in the two child classes of *ControlSystemPlayback*. *ControlSystemDevicePlayback* defines a method *registerControlDevice* that gives the playback a reference to one of the *ControlDevice* objects contained within the *ControlSystem*. Similarly, *ControlSystemSurfacePlayback* defines *registerControlSurface* to allow a *ControlSystem* to expose a *ControlSurface* reference to a playback object.

ControlSystemDevicePlayback maintains a mapping of *ControlDevices* to commands. For each device registered with a *ControlSystemDevicePlayback*, an entry is added to this mapping, and the command is registered with the ancestor class *InterpolatingPlayback*. The virtual method *update* is overridden in *ControlSystemDevicePlayback*. In playback mode, the *update* method of the parent class is called to drive the throttles and to obtain new values for the device commands. In recording mode, the *update* method records the throttle positions and the device commands at the recording sample rate. *ControlSystemSurfacePlayback* operates in a similar fashion, for the purpose of recording and playback of throttles and surface positions.

CockpitPlayback

CockpitPlayback provides a means to record the pilot's inputs during a simulation. In LaSRS++, aircraft are currently regimented into three categories: fighter, transport, and drop model. Each aircraft type has a specific cockpit interface that provides the aircraft with its inputs. The *CockpitPlayback* class is used by these three interfaces to record and playback a pilot's inputs.

A particular cockpit interface registers the variables that are to be recorded and played back just like *InterpolatingPlayback*. *CockpitPlayback* however does not have a sampling rate – it records data every frame. This is the only way to ensure that the dynamic response of a model during playback is identical to the original performance.

Summary

The design presented in this paper met all of the specified requirements. The functionality of *TrafficPlayback* alone demonstrates that the design is flexible enough to provide most of the required general capabilities. It allows a vehicle to specify a starting point within a playback file. It allows the simulation operator to start playback at any time. It repeatedly uses the same playback data to affect the motion of a positional model. It allows several vehicles to be driven from the same data file. *InterpolatingPlayback* demonstrates fulfillment of the remaining general requirements. It provides a means of recording data at integral divisions of the simulation frequency. It provides a means

of playing back data that was recorded at any frequency. The specific playback capabilities were fulfilled by the concrete, derived classes.

Each of the playback methods has been used in a number of experiments at NASA Langley Research Center (LaRC). For example, the *TrafficPlayback* class was used in the Low Visibility and Surface Operations (LVLASO) study in the Research Flight Deck (RFD) at LaRC. The project used the B757 model to study a means of improving the safety and efficiency of airport surface operations so that clear-weather capacities can be maintained during instrument weather conditions. The TrafficPlayback class was used to provide airport traffic during the study. The *TrafficPlayback* proved to be extremely efficient. When the B757 aircraft model is used by itself in the RFD it normally consumes on average 11 milliseconds of CPU time per frame. When traffic was added to the simulation using sixteen instances of the *TrafficPlayback* class, the average cpu time consumed per frame increased to only 13 milliseconds. Clearly each traffic model added to the simulation adds very little to the computational requirements of the framework. The *ControlSystemPlayback* class has also been used in several projects. For example, the Weather Accident Prevention (WxAP) project played back surface positions from flight test data to verify simulation model performance. The *CockpitPlayback* class is used on a daily basis by both developers and researchers to analyze the performance of both pilots and simulation models.

Object-oriented development in C++ allows the complexities associated with playback to be abstracted away behind concise interface definitions of the general purpose classes discussed in the present work. The diverse uses of *Playback* and *InterpolatingPlayaback* suggest that additional special purpose recording and playback features can be easily added to the framework. Development and maintenance efforts have been minimized by reuse of this robust foundation.

Although the design presented in this paper was originally designed to support flight simulation at NASA Langley Research Center, the design could be used in any object-oriented framework to heighten reuse and maintainability.

Bibliography

[1] Richard A. Leslie, et al. *LaSRS++ An Object-Oriented Framework for Real-Time Simulation of Aircraft.* Paper Number AIAA-98-4529, August, 1998.

[2] Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[3] Michael Madden, et al. *Constructing a Multiple-Vehicle, Multiple-CPU Using Object-Oriented C++*. Paper Number AIAA-98-4530, August, 1998.

[4] David Geyer, et al. *Managing Memory Spaces In An Object-Oriented Real-Time Simulation*, Paper Number AIAA-98-4532, August, 1998.